INSA Rennes – Département INFORMATIQUE

Système Expert pour Smartphones

Rapport de Conception

Romain Boillon, Olivier Corridor, Quentin Decré, Vincent Le Biannic, Germain Lemasson, Nicolas Renaud, Fanny Tollec

2010-2011

Projet sous la direction de Laurence Rozé

TABLE DES MATIERES

1. Introduction

Comme vous avez pu le voir lors de nos rapports précédents, notre projet, intitulé « Manage Yourself » consiste à développer une application de diagnostic et de prévention d'erreur pour les Smartphones Android et iPhone. Comme décrit dans notre rapport de spécification, notre projet est principalement découpé en deux parties. L'une est la partie mobile dans laquelle se trouvent le système expert en charge de la surveillance du Smartphone, le système de reporting qui génère les rapports d'erreurs et de bon fonctionnement, et le suivi utilisateur pour Android. L'autre partie est le serveur sur lequel s'effectue l'apprentissage des règles nécessaires pour le système expert ainsi qu'une interface administrateur.

Notre but est ici de vous présenter notre projet dans son état final, son fonctionnement et les différentes modifications qu'il a pu subir. Pour cela nous vous présenterons tout d'abord une description des modifications importantes de spécifications et de conceptions effectuées, puis nous vous présenterons un manuel utilisateur complet de notre logiciel, vous permettant de vous en servir sans connaître son fonctionnement interne. Nous vous décrirons enfin les différentes phases de tests effectués, puis notre bilan sur ce projet de toute une année.

2. Optimisation de spécification et de conception

Nous avons constamment cherché, durant la conception de notre logiciel, à optimiser au maximum la structure de celui-ci, afin de rendre le code plus clair, plus fonctionnel, et pour certaines parties moins spécifique au support d'utilisation, que ce soit pour Android, pour le simulateur ou pour le serveur. Des modifications ont donc été effectuées sur la spécification, et la conception des différentes parties de notre projet. Nous allons ici vous les présenter, afin de souligner les différences que vous pourriez remarquer par rapport aux descriptions des anciens rapports.

2.1. Partie Générale

La partie générale de notre code, commune à l'application Android, au simulateur et au serveur, a été de loin la partie la plus modifiée de notre logiciel. En effet, de par la récupération de ce code dans plusieurs parties, nous souhaitions une conception très générale, ne nécessitant que l'implémentation différente d'une ou deux fonctions pour chacun des supports. Nous avons de ce fait modifié plusieurs fois notre modélisation pour finalement réaliser cet aspect général. Toutes les parties spécifiques via l'héritage et la redéfinition de certaines méthodes peuvent ainsi utiliser cette partie.

La partie générale est composée de deux concepts importants sur lesquels se basent notre projet, à savoir la base de faits et la base de règles.

La base de faits nous permet de répertorier des faits de plusieurs types, à savoir Boolean, String, et Integer, ayant chacun un nom et une valeur. Nous avons décidé d'implémenter la classe Fait, avec une classe template pour assurer les différentes possibilités de type. Cette base de faits devant avoir une instance unique, nous avons également décidé d'utiliser un design pattern de type Singleton.

La base de règles quant à elle nous permet de répertorier une bibliothèque de règles, chacune se composant de conditions et d'actions. Chaque condition est composée d'un opérateur et de deux attributs: soit fixes, comme un integer, soit des faits présents dans la base de faits. Si les conditions sont respectées alors la règle est applicable et les actions sont exécutées.

Nous avons décidé d'utiliser un design pattern de type Fabrique pour implémenter notre base de règles. L'idée, comme évoqué précédemment, était de réaliser une implémentation générique utilisable pour le serveur, le simulateur et la partie de développement Android. De ce fait, une grande partie du code est similaire pour cette base de faits et règles. La fabrique, utile pour initialiser et remplir ces bases, sera la classe ou les différentes parties se différencient. L'idée est que ces différentes parties peuvent créer une autre fabrique héritant de la principale et peuvent ainsi redéfinir uniquement leurs méthodes spécifiques. Par exemple pour Android, les actions sont redéfinies puisqu'elles font appel à des fonctions propres au système pour exécuter ou tuer un programme.

2.2. Partie Android

2.2.1. Gestion des informations

Pour gérer les informations concernant les applications lancées et la mémoire utilisée, deux classes supplémentaires ont été créées :

- AppManager : Singleton chargé de l'actualisation des processus connus (afin de pouvoir ajouter de nouvelles applications dans la base de faits quand de nouveaux processus sont détectés), et des processus en cours d'exécution.

- MemoryManager : Singleton chargé de l'actualisation de la mémoire utilisée. A chaque actualisation, ce dernier lit le fichier « /proc/meminfo » contenant diverses informations concernant la mémoire du téléphone.

2.2.2. Récupération des logs des processus en cours d'exécution

Le lancement de fichiers exécutables afin de récupérer les journaux du téléphone s'est avéré ne pas être un choix idéal. En effet, d'une part, aucun exécutable n'est standard pour Android (certains présents sur une version peuvent ne pas être présents dans une autre, ou encore, peuvent être présents mais retourner des choses sous un format différent), ce qui n'assurait pas un fonctionnement de l'application sur tous les appareils, et d'autre part, la fonction Java Runtime.exec (utilisée pour lancer des fichiers exécutables) semblait provoquer des bugs, et son utilisation était déconseillé par beaucoup de développeurs Android. Un choix différent a donc été fait : utiliser le NDK (Native Development Kit) d'Android. Ce dernier est

l'équivalent de la JNI (Java Native Interface), permettant à un développeur d'utiliser du code en C afin de créer des bibliothèques qu'il est possible d'utiliser directement depuis le code Java.

Deux modules ont alors été développés à l'aide du NDK :

- Le premier permettant la manipulation des journaux du système (effacement / ouverture / lecture)

- Le second permettant d'obtenir la liste des identifiants des processus en cours

2.3. Partie iPhone

Pour ce qui concerne la partie pour l'iPhone, les quelques changements qui ont été opérés l'ont été à cause de la spécificité du langage (l'Objective-C), ou du SDK (Software Development Kit) de l'iOS.

Tout d'abord, la partie Monitoring de l'application, qui est constitué de la base de fait, a été amputé de nombreux faits à cause de la limitation du SDK d'iOS qui ne nous permet pas de récupérer ces informations. Ainsi nous n'avons pas pu récupérer les informations sur l'utilisation de l'appareil photo, de Bluetooth et du Wifi. A part ces modifications dues à des limitations du SDK, nous n'avons pas modifié la modélisation de la base de faits.

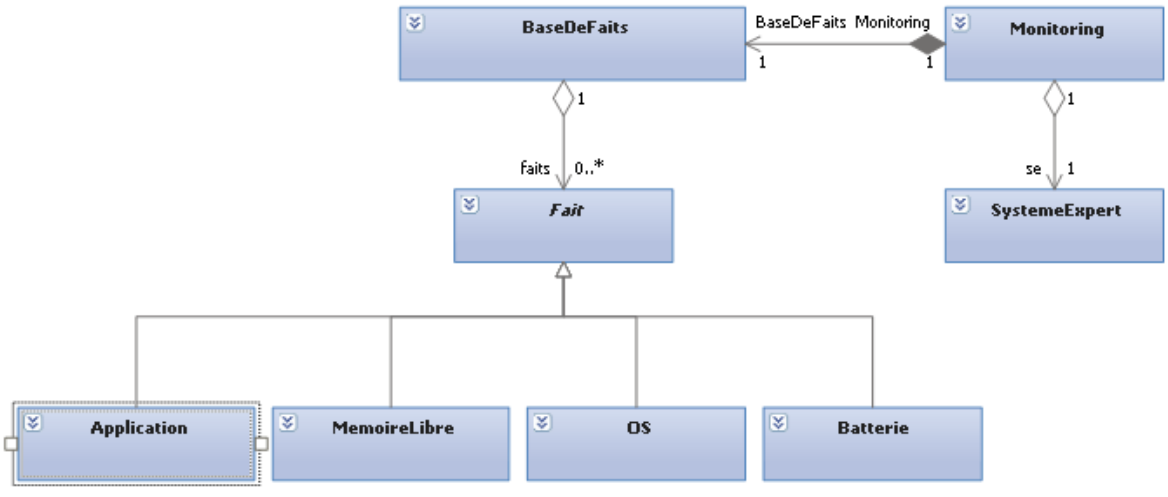

FIGURE 1 DIAGRAMME DE CLASSES DU MODULE DE MONITORING

C'est dans la partie du Système Expert que nous avons fait le plus de changement de modélisation. En effet suite à l'impossibilité de tuer ou de lancer une application grâce au SDK, nous n'avons pas pu implémenter ces actions, il ne reste donc que l'affichage de message. Un autre changement est celui des attributs dynamiques d'une comparaison: dans un premier

temps, nous pensions créer une classe d'attribut fixe par type (Int, String, Bool). Mais au moment de l'implémentation, nous nous sommes rendu compte que ces différentes classes étaient en tous points identiques, et nous avons donc implémenté qu'une seule classe représentant les attributs dynamiques de tous les types (classe AttributDyn). Nous avons aussi dû ajouter une classe (classe ReglesParser) pour lire les fichiers de règles XML afin de remplir la base de règles au lancement de l'application. Il était en effet préférable de séparer la partie qui remplit la base de règles à celle qui essaie d'appliquer les règles pour plus de lisibilité et de compréhension des différentes tâches.

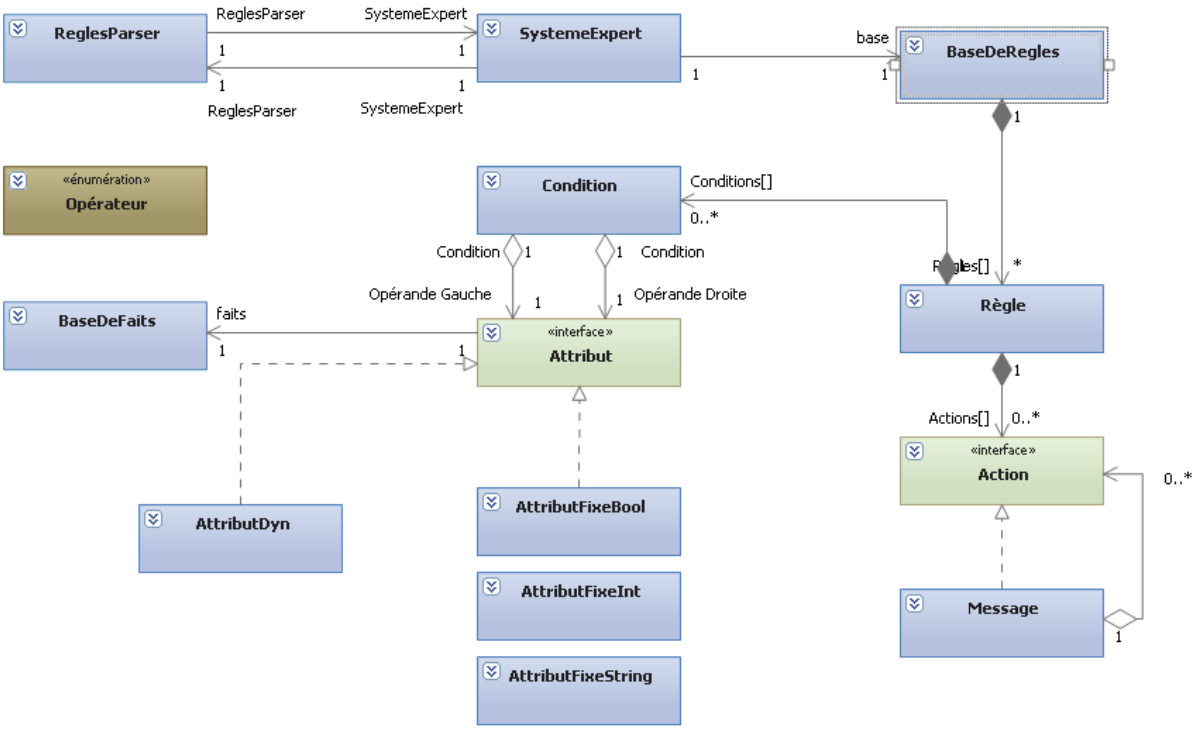

FIGURE 2 DIAGRAMME DE CLASSES DU MODULE SYSTEME EXPERT

Dans nos spécifications nous avions indiqué que notre application fonctionnerait en tâche de fond, le SDK d'iOS nous offrait 3 possibilités pour le faire: déclarer que l'application va utiliser la position GPS de l'iPhone, continuer une conversation en VoIP ou continuer de jouer de la musique lorsque l'application est en tâche de fond. Dans ces 3 cas il aurait fallu que notre application utilise les services respectivement de localisation, de VoIP ou d'audio de l'iPhone. Si nous avions utilisé le service de VoIP, cela aurait nécessité une connexion à un autre téléphone VoIP et aurait utilisé la connexion internet de l'utilisateur. Cette solution n'était donc pas viable. Le problème était similaire pour le service audio: cela empêchait l'utilisateur d'écouter de la musique en même temps que notre application, et nous aurions été obligé de lui faire écouter quelque chose. Il ne restait donc plus que le GPS. Mais celui-ci consomme énormément de batterie et le service de GPS n'est appelé que lorsqu'il y a un changement significatif de position,

donc si l'utilisateur ne changeait pas de position, notre application n'aurait pas fonctionné en tâche de fond. Cette solution n'était donc pas viable non plus. Notre application ne peut donc pas fonctionner en tâche de fond, l'utilisateur devra donc la mettre régulièrement au premier plan pour qu'elle fonctionne.

2.4. Partie Simulateur

La partie simulateur a été modifiée sur certains détails:

- l'horloge est maintenant gérée par le contrôleur qui se charge de l'incrémenter régulièrement. Les autres modules tels le reporting et le système expert ont néanmoins accès à sa valeur. (par exemple pour générer des rapports régulièrement). C'est le contrôleur qui simule l'ensemble de l'évolution du smartphone, il nous paraissait donc évident qu'il doive aussi simuler l'avancement de l'horloge.

- dans les fichiers d'entrée sur les attributs et les applications la balise "ID" ont été enlevée car elle était redondante avec le nom des attributs et applications.

Le diagramme de classe du simulateur n'a pas évolué.

2.5. Partie Serveur

2.5.1. Compilateur XML vers ARFF et concaténation de rapports

Le compilateur concaténant les rapports iPhone incomplets au format XML et créant des rapports complets (également au format XML) a été regroupé avec le compilateur prenant simplement des rapports XML et créant un unique fichier au format ARFF. Ainsi, il suffit de choisir lors du lancement quel mode on souhaite appliquer en passant un argument au programme.

Une recherche des types des différents attributs a également été ajoutée, permettant de repérer quels attributs ont un type numérique, et lesquels ont au contraire ont un type énuméré. Les attributs possibles ne sont donc plus intégrés directement dans le code, ce qui permet ainsi d'ajouter n'importe quel type d'attribut dans un rapport sans avoir à recompiler le compilateur.

2.5.2. Interface d'administration

L'interface d'administration, permettant d'ajouter des règles apprises par le système d'apprentissage à la base de règle en y adjoignant des actions associées, a été totalement refaite pour plusieurs raisons :

Il est nécessaire de gérer plusieurs systèmes, ne possédant pas forcément les mêmes faits surveillés, ni même les mêmes actions.

Les optimisations apportées à la structure générale imposent de les suivre également sur la partie interface administration.

Nous souhaitions offrir une interface plus pratique pour la gestion des règles, qui permet de ne pas connaître à l'avance les différents faits, actions possibles, ou encore les valeurs possibles.

D'autres améliorations ont été apportées comme :

- l'utilisation d'un seul et unique fichier projet
- la modularité de l'interface (paramétrable via un simple fichier XML)
- la proposition de choix possibles uniquement. Si l'utilisateur veut comparer un fait mémoire libre (entier), on ne lui proposera en partie droite qu'un entier ou un fait dont la valeur est un entier.

2.5.3. Une interface d'administration, plusieurs mobiles

On associe à une plateforme donnée un fichier XML, contenant de façon exhaustive les faits et actions supportés. Ici, on a 3 plateformes : Le smartphone Android, l'iPhone et le simulateur, mais l'utilisateur a la possibilité d'utiliser l'interface avec d'autres plateformes, simplement en créant un nouveau fichier XML. (Voir le manuel utilisateur). Cela permet aussi de ne pas avoir à modifier le code en cas de changement de la partie mobile.

Le XML contient donc les informations suivantes :

Les faits supportés par la plateforme, avec pour chaque fait : son nom, son type et éventuellement l'énumération de ses valeurs.

Les actions supportées par la plateforme, avec pour chacun : son nom, et ses paramètres avec leur type associé.

Voici un exemple de fichier XML. Ici, celui du simulateur.

```
<?xml version="1.0" encoding="UTF-8"?>
<ProjetManageYourself>
<basedeconnaissance>
    <regles>
        <regle id="1337">
```

```
            <conditions>
                <condition>
                    <valueType value="SUPEG" />
                    <dyna value="version" />
                    <fixe value="OS1" type="String" />
                </condition>
            </conditions>
            <actions>
                <action type="Execute" app="Update2" />
            </actions>
        </regle>
    </regles>
</basedeconnaissance>
<Parametres nom="Simulateur">
<attributs>
    <attribut nom="memoireVive" type="integer" min="0" max="1000" />
    <attribut nom="batterie" type="integer" min="0" max="100" />
    <attribut nom="typeplantage" type="enum">
        <valeur>none</valeur>
        <valeur>memoiresat</valeur>
        <valeur>applicrash</valeur>
        <valeur>lowbat</valeur>
    </attribut>
    <attribut nom="version" type="enum">
        <valeur>OS1</valeur>
        <valeur>OS2</valeur>
        <valeur>OS3</valeur>
        <valeur>OS4</valeur>
        <valeur>OS5</valeur>
    </attribut>
    <attribut nom="memoirePhysique" type="integer" min="1000" max="8000" />
    <attribut nom="APN" type="boolean" />
</attributs>
<actions>
    <action nom="Kill">
        <parametre nom="application" />
    </action>
    <action nom="Execute">
        <parametre nom="application" />
    </action>
</actions>
</Parametres>
</ProjetManageYourself>
```

2.5.4. Organisation des données

Nous avons maximisé la réutilisation du code de la partie générale, cela se traduit en premier lieu par les structures de données utilisées, que l'on peut voir sur le schéma ci-dessous :

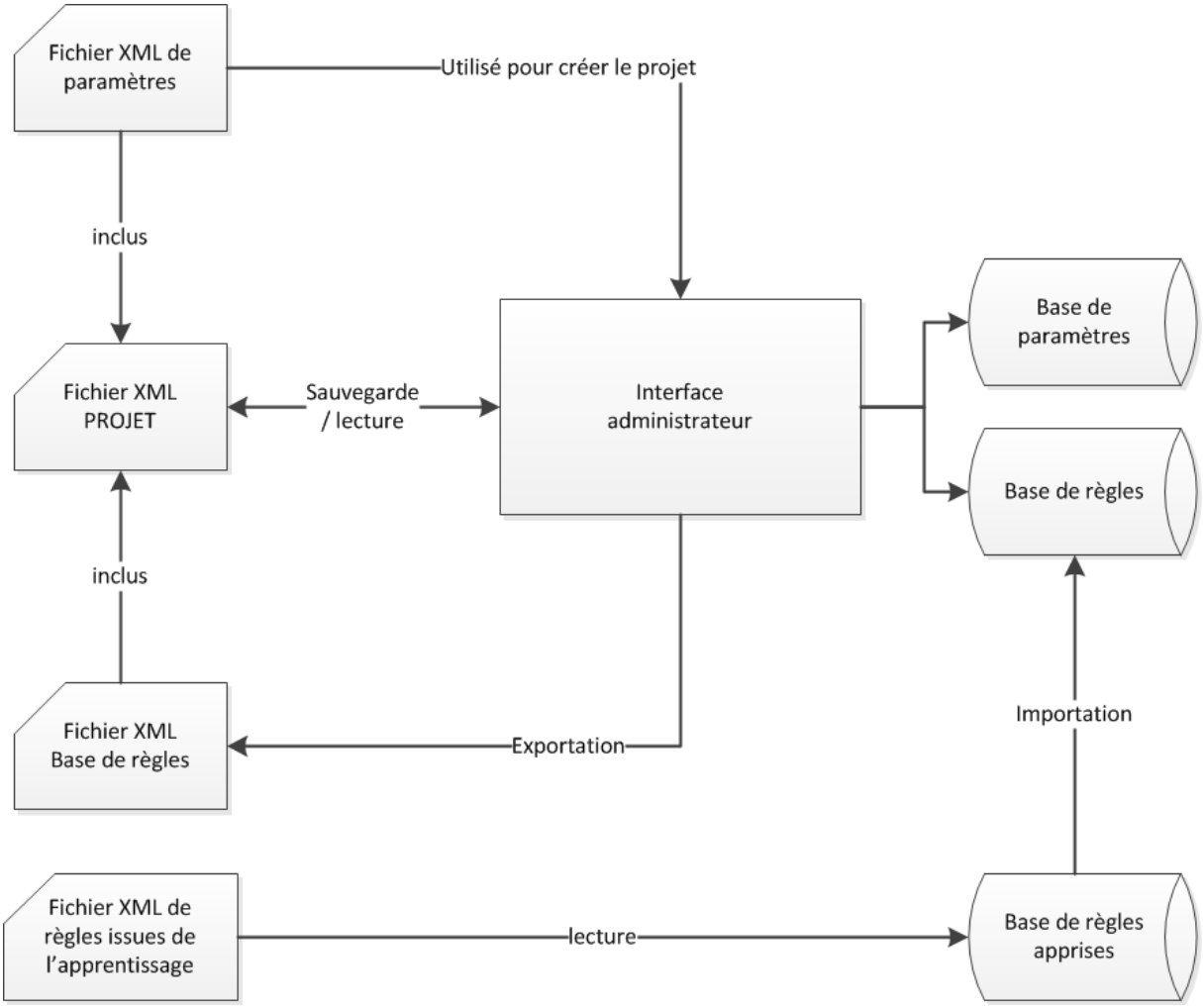

FIGURE 3 UTILISATION DES DIVERSES STRUCTURES DE DONNEES

Le serveur s'appuie également beaucoup sur le code de la partie général, tout en redéfinissant certaines méthodes pour subvenir au mieux aux besoins de l'interface. Mais il n'y a pas de code redondant ! Le diagramme de classe ci-dessous détaille cette structure.

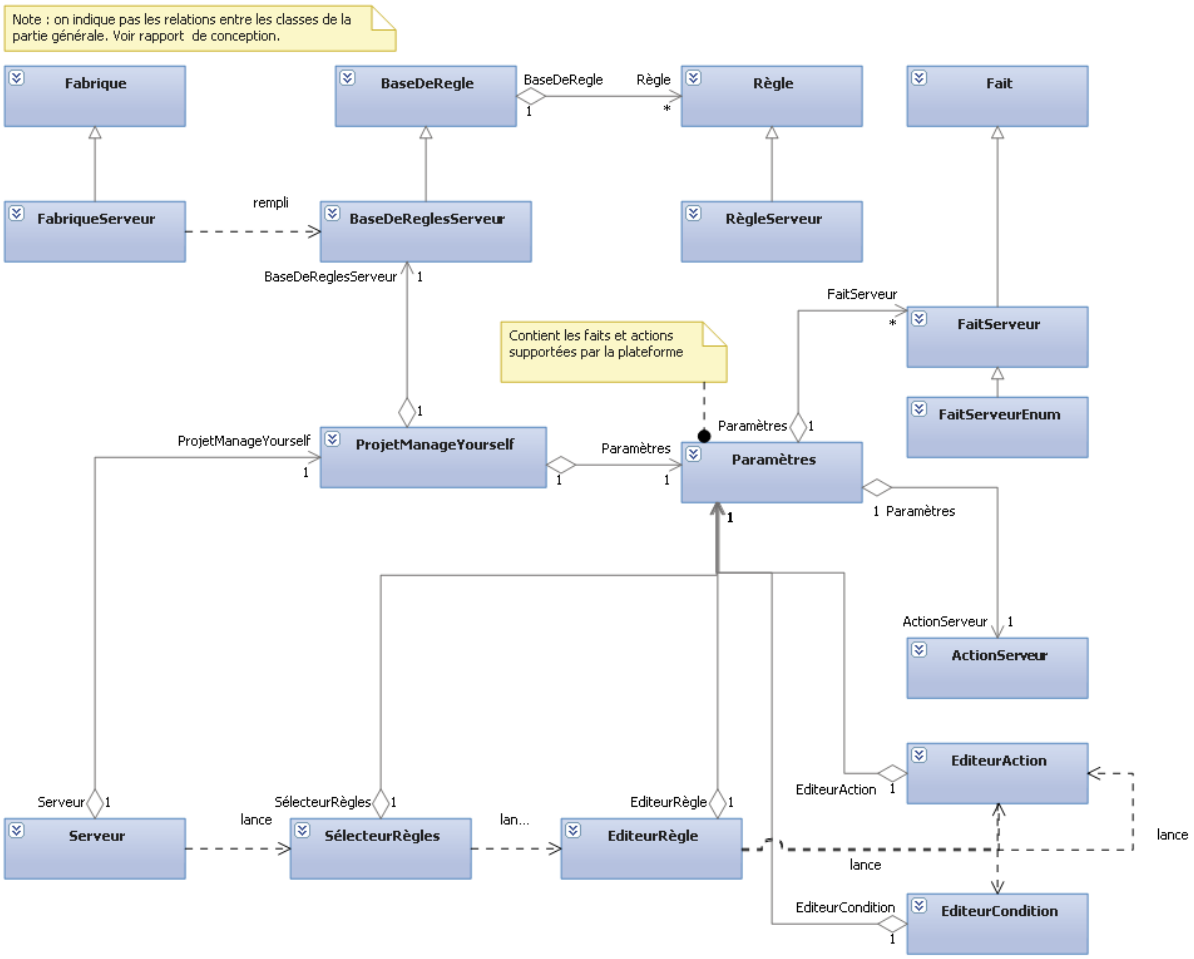

FIGURE 4 DIAGRAMME DE CLASSES DE LA PARTIE GENERALE

3. Manuel Utilisateur

Le manuel utilisateur que nous vous présentons ici a pour but de vous expliquer, en tant qu'utilisateur client de notre produit, son fonctionnement et son utilisation visible. Nous ne détaillerons pas ici le fonctionnement interne des différentes parties.

3.1. Installation et Utilisation

3.1.1. iPhone

L'installation de ManageYourself se fait uniquement en lançant l'application depuis Xcode sur l'iPhone connecté à l'ordinateur en usb, car il n'est malheureusement pas disponible sur l'Apple store

Les fichiers crées ou lus par ManageYourself sont situés dans le dossier propre à l'application. Dans ce dossier vous trouverez les rapports de bon fonctionnement générés, qui seront de la forme "rapport - XXXXXXXXXX.xml" (où XXXXXXXXXX est un entier représentant la date à laquelle le rapport a été créé), les rapports de bilan de règles ("bilanRegles - XXXXXXXXXX.xml"), et la base de règle ("baseDeRegles.xml") que vous pourrez consulter ou éditer. Ces rapports et la base de règles pourront être récupérés ou édités via iTunes et l'onglet "App", dans la section "Partage de fichier"

Après avoir lancé ManageYourself l'utilisateur arrive sur une interface simpliste, lui disant qu'il a bien lancé l'application et que le système expert est bien en marche.

FIGURE 5 ECRAN D'ACCUEIL DE L'APPLICATION IPHONE

Lorsqu'une règle est appliquée, une notification apparaît pour suggérer à l'utilisateur une action à faire pour éviter un crash du téléphone. Il peut l'accepter ou la refuser.

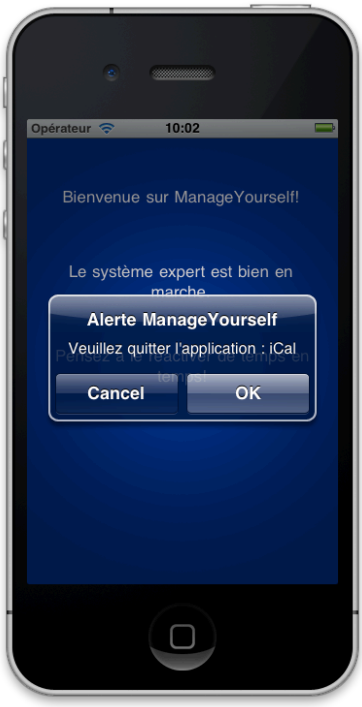

FIGURE 6 MESSAGE PROPOSANT L'APPLICATION D'UNE REGLE SOUS IPHONE

Lors de la fermeture de l'application, (J'espère, j'y suis pas encore arrivé) un rapport de règles est généré, rendant compte du nombre de fois que l'utilisateur a accepté ou non d'appliquer l'action de la règle.

L'application n'étant pas multitâche, l'utilisateur devra la remettre au premier plan pour qu'elle effectue ses traitements.

3.1.2. Android

L'installation de ManageYourself peut se faire, soit via l'interface de développement, soit en exécutant directement depuis le mobile le fichier .apk généré par le kit de développement Android.

Lors de son premier lancement, l'application vérifie si les répertoires et fichiers nécessaires sont déjà créés, et s'en charge si ce n'est pas le cas, en initialisant une base de règles vide.

Tous les fichiers concernant ManageYourself sont alors stockés dans le dossier « ManageYourself » présent sur le carte SD de l'appareil. Il est ainsi possible de modifier ou consulter la base de règles en éditant le fichier « basederegles.xml », et de récupérer les rapports (de bon et mauvais fonctionnement) générés par l'application en copiant les fichiers étant sous la forme « rapportXXXXX.xml » (où XXXXX est un entier représentant la data à laquelle le rapport a été créé).

Après le lancement de l'application, l'utilisateur accède à une interface où il peut choisir de démarrer ou arrêter le service de surveillance du téléphone.

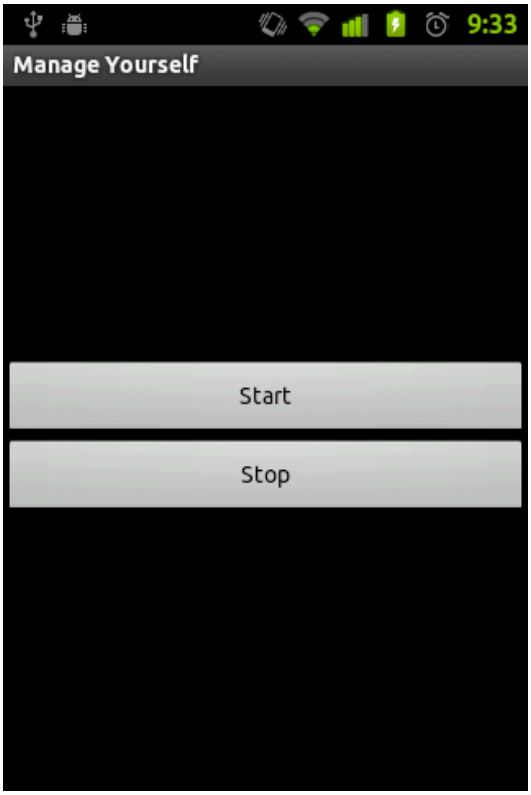

FIGURE 7 ECRAN D'ACCUEIL DE L'APPLICATION ANDROID

Lorsque le service est en cours d'exécution, une icône est affichée dans la zone de notification.

FIGURE 8 NOTIFICATION DE L'EXECUTION DU SERVICE

Lors d'un crash, une notification apparaît, et un rapport de crash est créé dans le dossier ManageYourself sur la carte SD du téléphone.

FIGURE 9 NOTIFICATION DE LA CREATION D'UN RAPPORT DE CRASH (1)

FIGURE 10 NOTIFICATION DE LA CREATION D'UN RAPPORT DE CRASH (2)

Certaines actions peuvent être effectuées arbitrairement, mais l'administrateur de la base de règles peut également choisir d'afficher un message à l'écran permettant de laisser le choix à l'utilisateur d'exécuter ou pas les actions proposées.

FIGURE 11 MESSAGE OFFRANT LE CHOIX A L'UTILISATEUR D'EFFECTUER OU NON DES ACTIONS

3.2. Génération de Rapports

Les rapports de bon fonctionnement sont générés à intervalle de temps régulier pour garder une trace de bon fonctionnement du système.

Les rapports de mauvais fonctionnement quant à eux sont générés lorsqu'un crash d'une application a lieu. Ils comportent plusieurs informations sur le moment du crash et l'état du téléphone.

Ces rapports sont envoyés au serveur afin d'enrichir la base de règles si de nouvelles conditions peuvent permettre un apprentissage.

3.3. Transmission au serveur

La transmission des rapports au serveur se fait manuellement, il faut pour cela copier les rapports étant situés dans le répertoire “ManageYourself” sur la carte SD du smartphone Android, ou pour iPhone à l’adresse suivante pour les rapports de mauvais fonctionnement :

```
C:\Users\<user>\AppData\Roaming\AppleComputer\Logs\CrashReporter
\MobileDevice\<device>\
```

et via iTunes et son onglet App pour les rapports de bon fonctionnement.

3.4. Serveur et déduction de règles

3.4.1. Génération des rapports

Les rapports de bon fonctionnement (iPhone et Android) sont générés périodiquement par l’appareil et contiennent toutes les données que contient la base de faits au moment de leur création.

Les rapports de bug Android sont quant à eux également générés par l’application quand un crash est détecté. Ils ne nécessitent aucune manipulation supplémentaire car contiennent les mêmes informations qu’un rapport de bon fonctionnement.

En ce qui concerne les rapports de bug de l’iPhone, ils sont générés à partir des crashs logs fournis par le téléphone qui sont créés lorsqu’une application plante, ou que le téléphone plante à cause d’une saturation mémoire ou d’une batterie déchargée. Ces crashs logs sont accessibles à l’utilisateur lorsqu’il synchronise son iPhone avec iTunes, ils sont alors disponibles dans des fichiers .crash à l’adresse suivante :

```
C:\Users\<user>\AppData\Roaming\AppleComputer\Logs\CrashReporter\MobileDevi
ce\<device>\
```

Afin de générer des rapports de mauvais fonctionnement en XML à partir des crashs logs, l’utilisateur a à sa disposition un script PERL qui s’en occupe. Pour cela il faut lancer le script avec la commande suivante :

```
    perl LectureiPhone <Dossier>
```

Le paramètre <Dossier> correspond au dossier dans lequel sont situés les crashs logs générés par l’iPhone et où vont être créé les rapports au format XML.

3.4.2. Apprentissage

Une fois les rapports générés et récupérés, ces derniers doivent être convertis dans un format lisible par Weka, qui se chargera d'appliquer l'algorithme d'apprentissage.

Pour cela, on passe par un outil prenant en entrée des rapports XML, et les convertissant au format XML. Ce dernier peut également être utilisé pour concaténer des rapports de bon et mauvais fonctionnement iPhone afin de trouver les attributs manquants dans les rapports de crash.

Pour l'utiliser, il faut tout d'abord placer l'ensemble des rapports dans le répertoire "reports" du compilateur XmlToArff. Ces derniers ne nécessitent pas un nommage particulier, le compilateur les lisant tous avant de les traiter par ordre chronologique (la date de création étant indiquée dans chaque rapport). Le lancement s'effectue alors, soit en utilisant directement les deux scripts de lancement "run_concatenation.bat" et "run_xmlarff.bat" effectuant respectivement la concaténation et la compilation au format ARFF, soit directement en ligne de commande en respectant la syntaxe suivante :

```
java -classpath ./bin;./jdom.jar
com.manageyourself.reportscompiler.Compilateur <MODE>
```

Le paramètre <MODE> peut être le suivant :

-c : Lancement en mode concaténation de rapports

-x : Lancement en mode compilateur XML vers ARFF

Si le paramètre n'est pas spécifié, l'outil se lance en mode concaténation.

Dans le premier cas (lancement en mode concaténation), le résultat de l'opération est un ensemble de fichiers xml créés dans le répertoire "output" du compilateur. Ces nouveaux rapports peuvent alors, si nécessaire, être réutilisés en mode compilateur XML vers ARFF.

Dans le second cas (lancement en mode compilateur XML vers ARFF), un unique fichier output.arff est créé dans le répertoire "output". Ce dernier peut alors directement importé dans Weka.

3.4.3. Interface d'administration

3.4.3.1. Création de projet

Il suffit d'indiquer le type du projet (iPhone, Android, Simulateur, ou personnalisé auquel cas il faudra fournir un fichier XML de paramètres), puis d'indiquer le nom et l'emplacement du projet.

Le nouveau fichier projet est créé et il est alors possible d'éditer les règles.

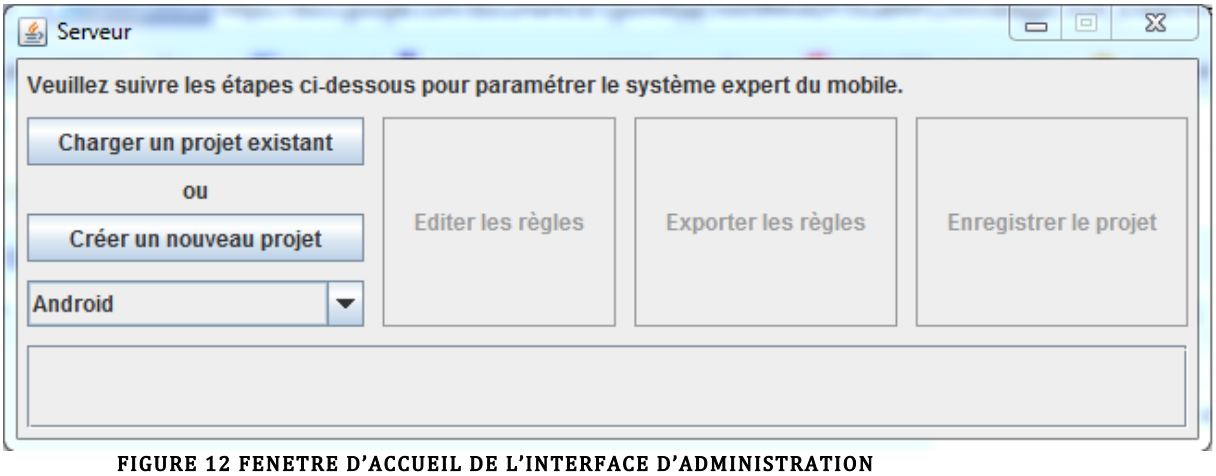

FIGURE 12 FENETRE D'ACCUEIL DE L'INTERFACE D'ADMINISTRATION

3.4.3.2. Charger un projet

En cliquant sur "charger" puis en indiquant le fichier du projet, il est chargé, les paramètres sont également automatiquement détectés. Inutile d'indiquer le type du projet au préalable.

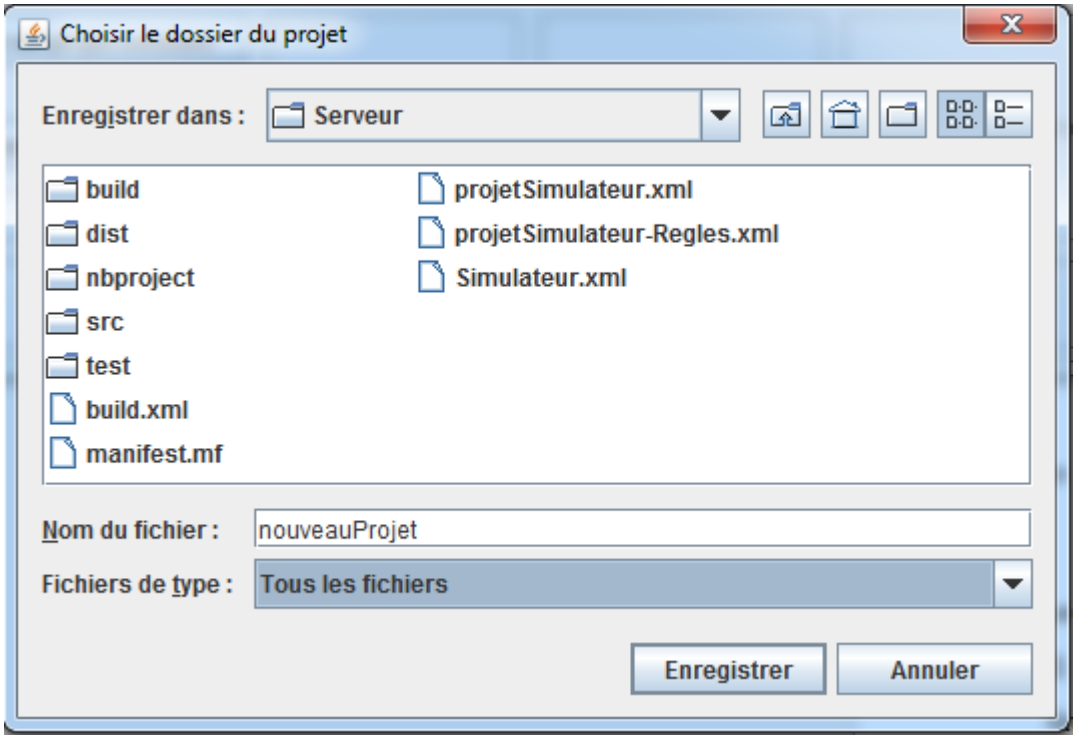

FIGURE 13 ENREGISTREMENT D'UN PROJET

3.4.3.3. Modifier la base de règles

En cliquant sur "charger" puis en indiquant le fichier du projet, il est chargé, les paramètres sont également automatiquement détectés. Inutile d'indiquer le type du projet au préalable.

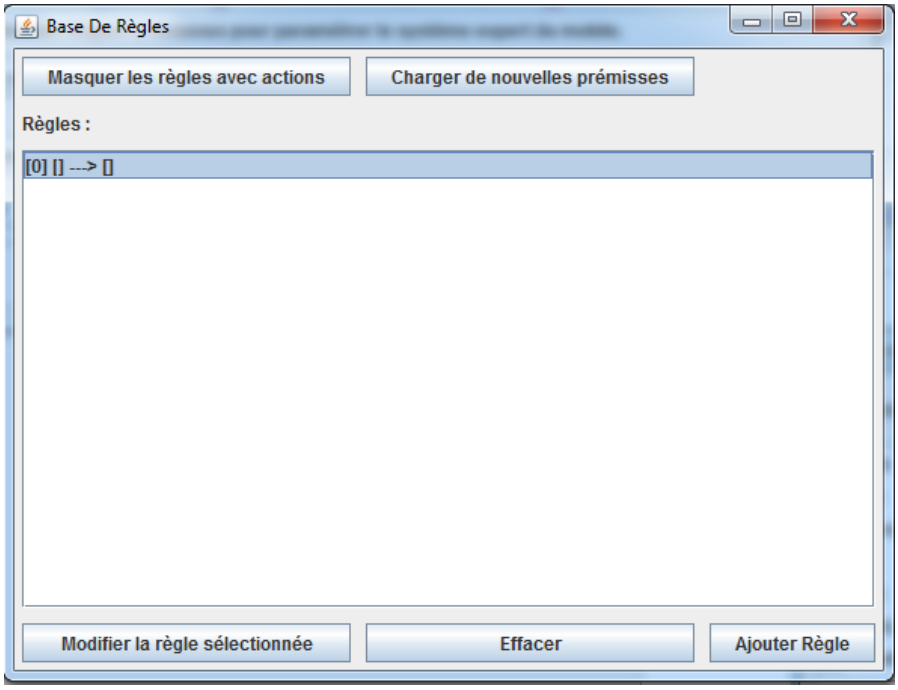

FIGURE 14 MODIFICATION DE LA BASE DE REGLES

3.4.3.4. Modifier une règle

Dans le cas d'un ajout ou d'une modification, une fenêtre s'ouvre, détaillant la règle :

- Ses conditions
- Ses actions

On a alors la possibilité pour les actions et conditions, d'en supprimer, d'en modifier ou d'en ajouter.

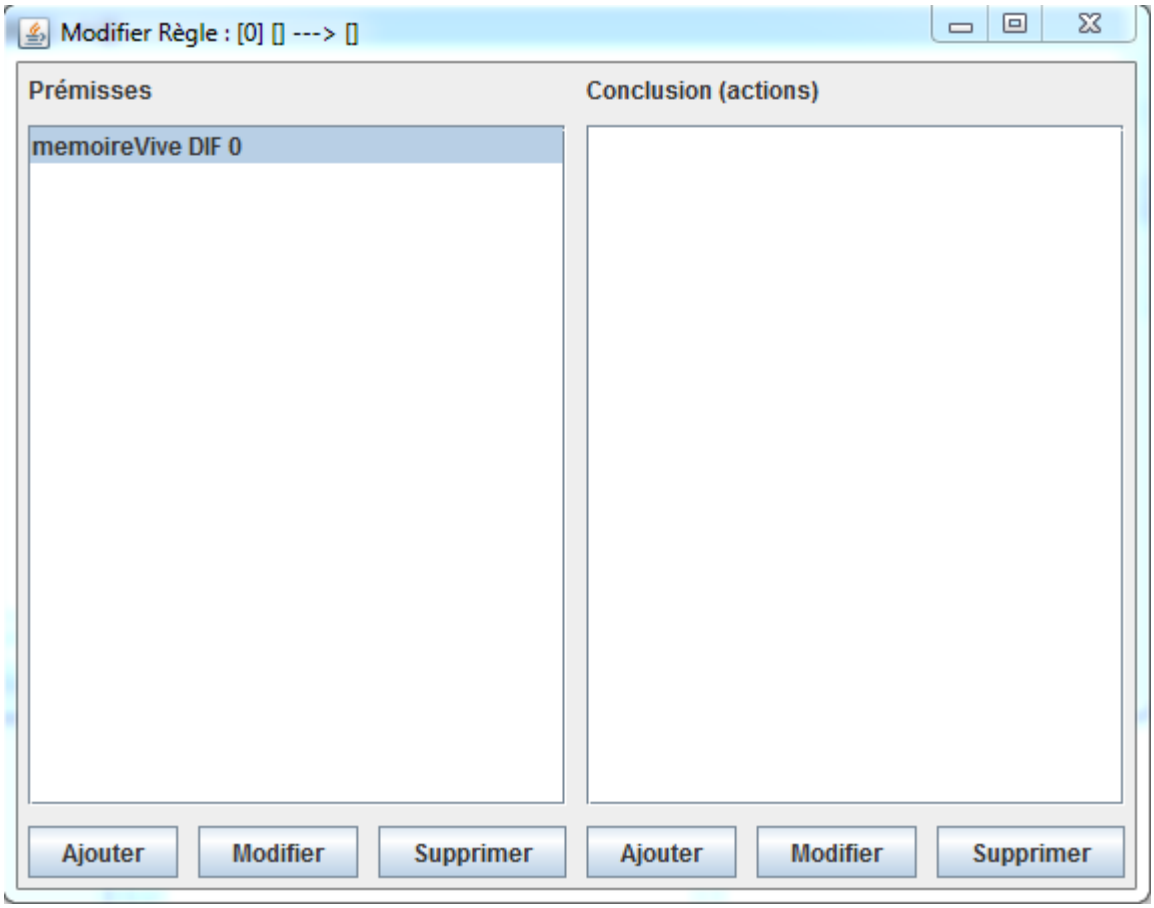

FIGURE 15 MODIFICATION D'UNE REGLE

3.4.3.5. Modifier une condition

Encore une fois, l'application nous propose les attributs possibles, et lorsque le premier attribut est choisi, ne restent en proposition dans la partie droite, que les attributs compatibles par leur type.

Si l'attribut gauche a plusieurs valeurs énumérées possibles, elles sont ajoutées en partie droite, ce qui évite à l'utilisateur de les connaître par cœur (voir deuxième capture).

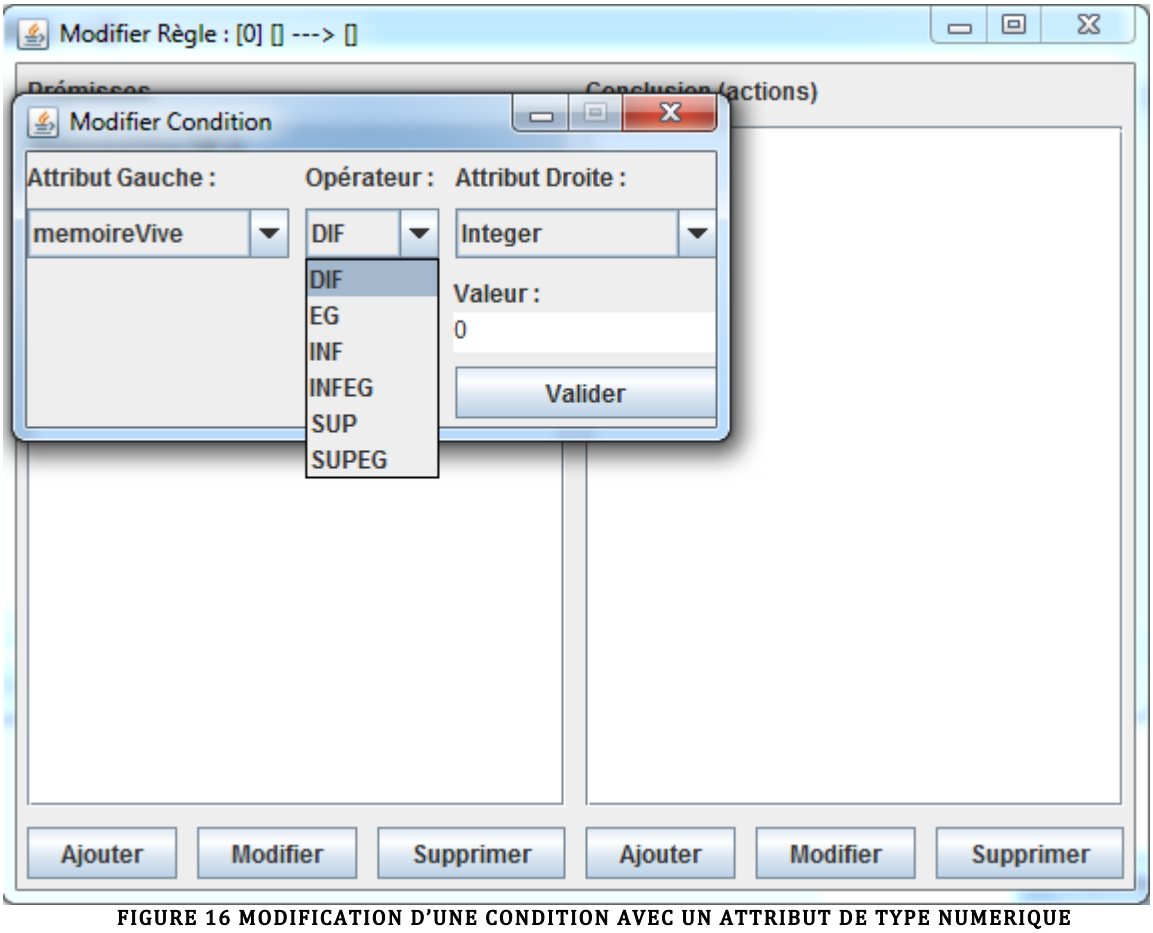

FIGURE 16 MODIFICATION D'UNE CONDITION AVEC UN ATTRIBUT DE TYPE NUMERIQUE

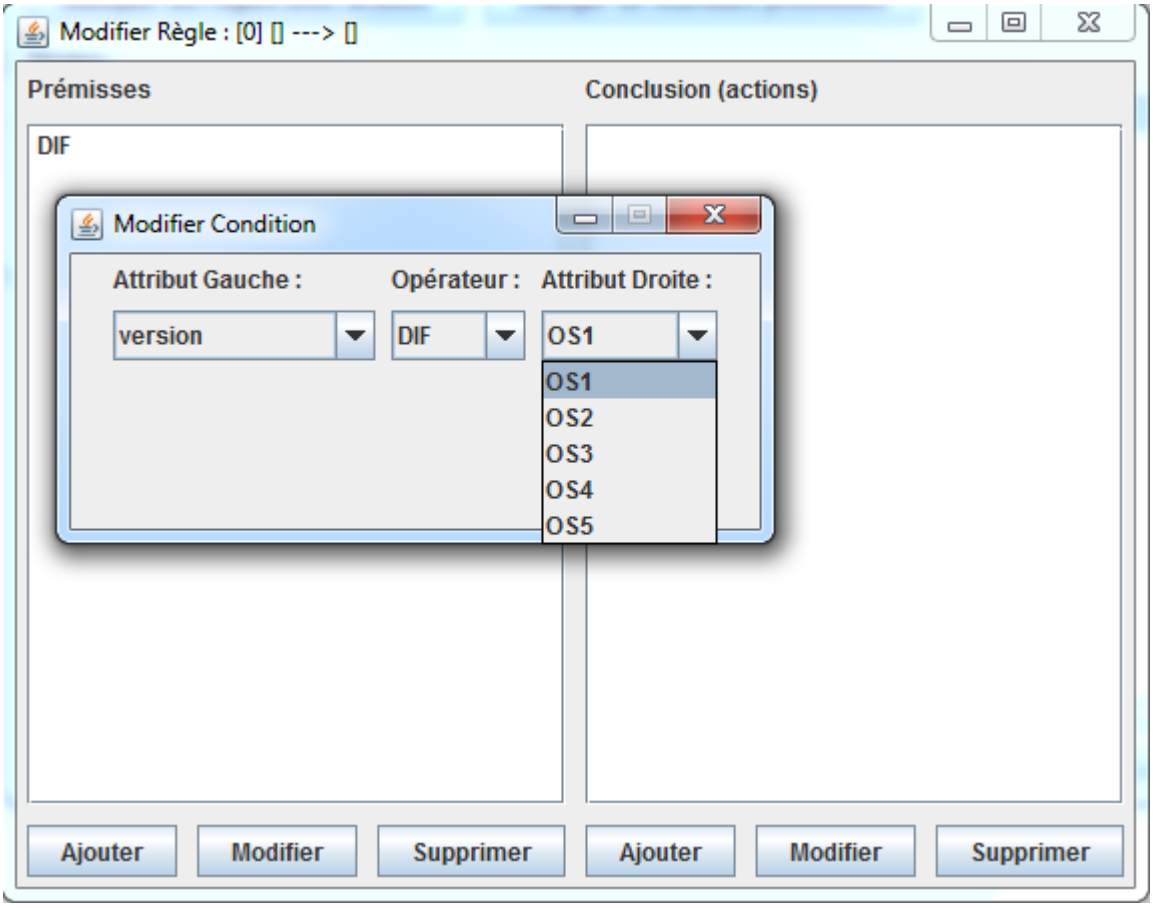

FIGURE 17 MODIFICATION D'UNE CONDITION AVEC UN ATTRIBUT DE TYPE ENUMERE

3.4.3.6. Modifier une action

Les actions possibles sont présentées. Si l'action est de type message, on peut voir les actions filles (on rappelle que les messages emploient le design pattern composite).

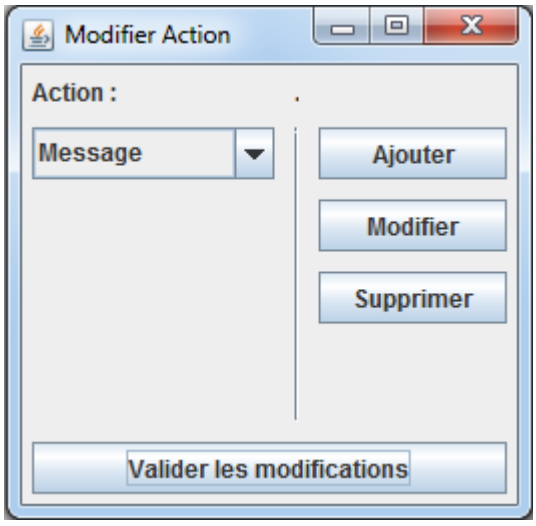

FIGURE 18 MODIFICATION D'UNE ACTION

On peut ainsi composer à l'infini la hiérarchie des messages. Des champs permettant de remplir les paramètres de l'action sont automatiquement affichés en fonction de l'action sélectionnée.

3.4.3.7. Exporter la base de règles

Le bouton exporter la base de règles de la fenêtre principale permet de choisir un nom de fichier qui sera rempli avec le contenu de la base de règle au format XML.

3.4.3.8. Sauvegarder le projet

Sauvegarde les paramètres et la base de règle dans un unique fichier XML (celui indiqué à l'ouverture ou à la création).

3.5. Application de nouvelles règles

Les nouvelles règles sont transmises au téléphone via le fichier XML de la base de règles. Le système doit alors être redémarré pour prendre en compte les différentes modifications apportées. Après son lancement, il recommencera alors à actualiser à intervalle réguliers la base de faits, et à vérifier si les règles doivent ou non être appliquées.

3.6. Utilisation du simulateur

3.6.1. Lancement

Le simulateur se présente sous la forme d'un fichier .jar qui doit être lancé en ligne de commande, avec plusieurs paramètres:

- 5 fichiers .xml correspondants aux entrées du simulateur
- la destination des fichiers de sortie du simulateur
- la durée de la simulation en jours simulés

Arguments possible :

-att [attribut.xml]

-attSim [attributSimu.xml]

-app [applications.xml]

-re [regleTel.xml]

-ba [basederegle.xml]

-rapp [Rapport/]

-d [durée en jour ex : 2]

3.6.2. Format des fichiers d'entrée du simulateur

Ces fichiers sont au nombre de 5

- fichier de description des applications: -app

```
<applications>
   <application nom="AppliA" >
       <frequence min="180" max="240" />
       <duree min="10" max="20" />
       <effets>
           <effet type="ajout_temporaire">
               <target nom="memoireVive" />
               <value valeur="150" />
           </effet>
           <effet type="ajout_periodique">
               <target nom="batterie" />
               <interval valeur = "10" />
               <value valeur="1" />
           </effet>
       </effets>
   </application>
   <application nom="Recharge" >
       <frequence min="500" max="600" />
    …
</applications>
```

Une application est représentée par son nom.

La fréquence comprend une valeur *min* et *max* et signifie que l'application se lance aléatoirement entre ces deux valeurs.

La durée de l'application comprend une valeur *min* et *max* et signifie que l'application tournera pendant une durée comprise entre ces deux valeurs.

Il existe quatre types d'effets pour les applications:

- "*ajout_temporaire*": pendant que l'application est lancée un attribut (*target*) de type *numeric* sera incrémenté d'une valeur *value* donnée (ex: memoireVive). Pour le décrémenter il suffit de préciser une valeur négative. Une fois l'application fermée cet effet est inversé.
- "*ajout_periodique*": pendant que l'application est lancée un attribut de type *numeric* est incrémenté a chaque intervale de temps précisé. Ici encore, pour décrémenter l'attribut il suffit de préciser une valeur négative.

- "*ajout_permanent*": au lancement de l'application l'attribut cible (*target*) de type *numeric* est incrémenté de la valeur précisée. Contrairement à l'effet temporaire elle n'est pas réinitialisée à la fermeture de l'application.
- "*setter*": permet de fixer directement la valeur d'un attribut (*numeric* ou *enum*) à la valeur précisée.

- fichier de description des attributs : -att

```
<attributs>
   <attribut nom="memoireVive" type="numeric" min="0" max="1000" />
   <attribut nom="batterie" type="numeric" min="0" max="100" />
   <attribut nom="typeplantage" type="enum" >
       <valeur>none</valeur>
       <valeur>memoiresat</valeur>
       <valeur>applicrash</valeur>
       <valeur>lowbat</valeur>
   </attribut>
   <attribut nom="version" type="enum" >
   …
</attributs>
```

- Un attribut est représenté par son nom.
- Il existe deux types d'attributs:
    - *numeric*: compris entre une valeur *min* et *max* (ex: memoireVive)
    - *enum*: prend les différentes valeurs décrites (ex: typeplantage)

- fichier complémentaires sur les attributs (propre au simulateur): -attSim

```
<attributs>
   <attribut nom="memoireVive"  default="200" reset="true" />
   <attribut nom="batterie" default="100"  reset="false" />
   <attribut nom="typeplantage" default="none" reset="true"/>
   <attribut nom="version" default="OS1" reset="false"/>
   <attribut nom="memoirePhysique" default="1000" reset="false"/>
    ...
</attributs>
```

- *"default"* est la valeur par défaut prise par l'attribut au démarrage du smartphone.
- "*reset*" est à *true* si l'attribut est réinitialisé (à *default*) après un reboot de l'appareil, et à *false* si il conserve sa dernière valeur avant le reboot.

- fichier des règles de plantage du simulateur: -re

```
<basedeconnaissance>
   <regles>
       <regles>
        <regle id = "42">
            <conditions>
                <condition>
                    <valueType value = "SUP" />
                    <dyna value = "memoireVive" type="num" />
                    <fixe value = "400" type="Integer" />
                </condition>
                <condition>
                    <valueType value = "EG" />
```

```
                    <fixe value = "true" type="Boolean" />
                    <dyna value = "AppliC" type="app" />
                </condition>
            </conditions>
            <actions>
                <action type = "reset" param="memoiresat" />
            </actions>
        </regle>
    ...
   </regles>
</basedeconnaissance>
```

- Une règle est composée d'une série de conditions et d'une série d'actions. Elle se lit de la manière suivante: Si les conditions sont vraies alors faire les actions.
- Une condition est constituée de deux valeurs, une fixe et une dynamique ayant chacun un type (*String*, *enum*, *boolean*, *app*, *num*), ainsi que d'un opérateur *valueType* qui peut être *EG* (égal), *SUP* (supérieur), *INF* (inférieur) .
- Il existe qu'une action possible, c'est le redémarrage du smartphone virtuel. On peut indiquer en paramètre le type de plantage.

- fichier de la base de règles du simulateur : -ba

3.6.3. Format des fichiers de sortie du simulateur

Le simulateur rend en sortie deux types de fichiers:

- un fichier log qui retrace le déroulement de la simulation ainsi que les erreurs éventuelles.
- une série de rapports numérotés dans leur ordre d'apparition (leur nombre dépend du paramètre durée de la simulation). Ces rapports sont générés régulièrement dans le répertoire indiqué -rapp. Un rapport peut être soit de bon fonctionnement, soit de mauvais fonctionnement.

```
<rapport>
<batterie value="100" />
<memoireVive value="200" />
<typeplantage value="none" />
<memoirePhysique value="1000" />
<version value="OS1" />
<Application name="Reharge" value="notrunning" />
<Application name="AppliC" value="running" />
<Application name="AppliA" value="notrunning" />
<Application name="FuiteMemoire" value="notrunning" />
<Application name="Update2" value="notrunning" />
<Crash/>
</rapport>
```

Un rapport est constitué de la liste des attributs présents au moment de sa création et de leur valeurs respectives, ainsi que de la liste des application et leur état. La balise crash indique si le rapport et de bon ou de mauvais fonctionnement. (pas de balise crash si bon fonctionnement).

4. Tests

Grâce à la communication en XML entre les différents composants présents dans notre projet, nous avons pu entamer la conception des différentes parties de façon indépendantes les unes des autres, et donc surtout, simultanément. Seul le code du système expert était commun à plusieurs parties. Et seul le serveur a été entamé une fois le reste quasi fonctionnel.

Nos séries de tests se décomposaient donc en deux parties, suivant une logique de test classique : d'une part les tests des différentes parties indépendamment les unes des autres (test unitaires). Et d'autre part les tests de bon fonctionnement des différentes parties réunies (test d'intégration et de recettes).

Une fois ces derniers tests effectués, notre logiciel est, en théorie, totalement fonctionnel.

4.1. Tests unitaires

Les tests unitaires ont été réalisés avec JUnit sur chacune des fonctions, classe par classe, afin de tester leur bon fonctionnement. Cela nous a permis d'assurer une efficacité et une absence d'erreur à l'intérieur de notre code.

Pour réaliser ces tests, nous avons défini des règles, ainsi que des conditions et des actions simples à comprendre, et facilement implémentables dans un premier temps. Chacune des fonctions de nos classes se servant de ces exemples simples pour être validées, il était facile de voir les erreurs éventuelles et de les corriger. Une fois le résultat obtenu sur ces simples exemples, nous avons décidé de créer des règles plus complexes, composées de conditions plus élaborées et de multiples actions différentes dans le but de confirmer nos premiers résultats.

Une fois ces tests unitaires parfaitement réalisés, nous avons pu entamer sereinement les tests d'intégrations en liant les différentes parties entre elles.

4.2. Tests d'intégration

Comme cela est visible tout au long de nos rapports, notre projet est très nettement séparé en plusieurs parties distinctes. Assez distinctes pour ne même pas être codées dans le

même langage, sans que cela ne pose le moindre problème. Les tests d'intégrations ont donc ici une importance capitale pour le succès de notre projet, car ils assurent au final une réunion correcte de logiciels presque indépendants en un gros logiciel, beaucoup plus complexe, répondant au cahier de charge de notre projet d'année.

Notre choix fait dès la pré étude de communiquer entre les différentes parties grâce à un langage XML facilite cette communication, car alors de nombreuses parties ont pu être testées grâce à des fichiers XML d'entrée/sortie type, et tant que les fichiers XML correspondent, ne peuvent que communiquer correctement.

4.3. Tests de recette

Une fois arrivé à cette phase de test, nous sommes sûr que chaque partie indépendantes de notre logiciel, fonctionnent correctement. Chaque fonction, chaque partie du code, et chaque communication a été testée pour tous les cas d'utilisations hypothétiquement possibles.

Il ne nous reste donc plus qu'à faire les tests de recette de notre logiciel. Il nous faut tester, à travers des exemples types, si notre projet fonctionne dans sa globalité. A la conclusion de cette phase de test, notre logiciel doit fonctionner d'une façon identique à celle proposée dans le manuel utilisateur, sans erreurs ni manipulations supplémentaires à effectuer.

Nous devons donc tenter de trouver quelques exemples types du fonctionnement de notre projet, qui, sans être totalement exhaustifs, doivent au moins tester de façon approfondie le logiciel proposé.

Nous testerons alors, sous Android et sous iPhone, le système de création de règles, de l'activation du module de reporting sur le mobile, jusqu'à la récupération et l'application des règles trouvées sur le Smartphone.

On peut tester les exemples :

Sous Android : Un logiciel « probleme » est installé, générant un crash au bout de quelques secondes lorsque l'application « facebook » est allumée. L'activation des deux applications est faite à répétition. On récupère les rapports de bon/mauvais fonctionnement, compile et génère les règles sur weka, puis sélectionnons sur les serveurs la règle idéalement liée à notre problème, la liant avec l'action « kill probleme ». Nous insérons cette règle dans le

Smartphone, et testons de nouveau cette situation. L'application « probleme » doit automatiquement se fermer si facebook est ouvert.

On pourra tester ensuite une situation similaire, résultant au final à l'affichage d'un message, nous proposant de oui ou non fermer les deux applications, puis rouvrir l'application facebook, afin de tester le fonctionnement des différentes actions implémentés de notre logiciel.

Test sur Android de mauvaise règle : Aucun logiciel spécifique n'est installé. On récupère les rapports de bon et mauvais fonctionnement tel que trouvés, générons des règles et les installons. Une fois les règles testées, nous ne souhaitons plus les avoir sur le Smartphone (mauvaises règles). Nous retournons donc sur le serveur afin de supprimer ces règles de la base de règle de notre Smartphone.

Test de bon fonctionnement iPhone : Les mêmes tests peuvent être appliqués sur notre application iphone, en tenant compte des restrictions dues à la machine : aucune actions concrètes ne seront appliquées, mais à chaque situation problématique un message décrivant l'erreur apparaitra sur l'écran d'iphone.

De même, nous testerons, dans un deuxième temps, la suppression d'une erreur inintéressantes via le serveur.

5. Conclusion

A l'heure actuelle, les points suivants de notre projet sont fonctionnels:

Premièrement, la partie de lecture et d'écriture de fichiers de règles est opérationnelle. Les rapports peuvent être lus et écrits de manière totalement fonctionnelle. D'autre part, le système expert est capable de notifier les erreurs, et son module de génération de rapports fonctionne parfaitement en cas de notification.

Les systèmes de monitoring et d'application des règles, sur chacun des Smartphones étudiés, fonctionnent correctement. Due aux restrictions de l'iPhone, les règles applicables sur iPhone sont extrêmement réduites, mais le système en général marche aussi bien, voire mieux que ce qui était initialement prévu.

La déduction des règles à appliquer grâce à weka est correcte, et applicable pour notre projet.

Une nouvelle interface a été créée pour le serveur, répondant d'avantage à nos besoins. Le tout fonctionne correctement.

En conclusion, nous avons passé plus de temps que prévu sur les phases de développement et de tests du aux changements de modélisation. Cependant, malgré ces changements, une grande partie du projet est fonctionnel. Nous pouvons donc dresser, au final de cette année, un bilan positif de notre travail fourni.